HANSER

Leseprobe

Dirk Louis, Peter Müller

Android

Der schnelle und einfache Einstieg in die Programmierung und Entwicklungsumgebung

ISBN (Buch): 978-3-446-44598-7

ISBN (E-Book): 978-3-446-45112-4

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44598-7

sowie im Buchhandel.

Inhalt

Vorwort

Willkommen in der Android-Welt! Seitdem sich der Touchscreen als Standardoberfläche von Mobilfunktelefonen etabliert hat und vor Kurzem noch völlig unbekannte Features wie GPS-Empfänger und Lagesensor zur Standardausstattung gehören, gibt es kein Halten mehr: Jede Woche erscheinen neue Android-basierte Geräte und die Zahl der verfügbaren Apps im Android Market explodiert geradezu.

Wenn auch Sie dazugehören wollen, wenn Sie nicht bloß Anwender sein möchten, sondern daran interessiert sind, eigene Ideen in Apps umzusetzen – sei es zum Spaß oder auch vielleicht als Einstieg in eine Existenz als selbstständiger Software-Entwickler –, dann kann Ihnen dieses Buch einen guten Einstieg (und ein bisschen mehr) in die Welt der App-Programmierung für Android-Systeme bieten.

Vorkenntnisse und Anforderungen

Wir wollen nichts beschönigen. Die Anforderungen an Android-Programmierer sind hoch. Doch mithilfe dieses Buchs und ein wenig Ausdauer und Mitdenken sollten Sie die größten Hürden meistern können.

Sehen wir uns dazu einmal an, welche Fähigkeiten ein Android-Programmierer besitzen muss und inwieweit Ihnen dieses Buch helfen kann, diese Fähigkeiten zu entwickeln.

- *Gute Kenntnisse der Programmiersprache Java*
  Sie erfüllen diesen Punkt nicht? Kein Grund zur Panik, aber lesen Sie unbedingt den nachfolgenden Abschnitt zum „idealen Leser".
- *Umgang mit der integrierten Entwicklungsumgebung (IDE) Android Studio.*
  Alles, was Sie zum Umgang mit Android Studio im Allgemeinen wie auch im Hinblick auf die Erstellung von Android-Apps wissen müssen, lernen Sie in diesem Buch. Zusätzlich finden Sie am Ende des Buchs einen eigenen Anhang zu Android Studio, wo die wichtigsten Aufgaben noch einmal zusammengefasst sind (inklusive eines kleinen Wörterbuchs, das Lesern, die im Englischen nicht so versiert sind, die Eingewöhnung in die durchweg englische Benutzeroberfläche erleichtern soll).
- *Einsatz verschiedener Hilfsprogramme wie HierarchyViewer, Debugger und Emulator.*
  Insbesondere der Emulator ist für die Entwicklung von Apps unerlässlich, da Sie mit seiner Hilfe unterschiedlich ausgestattete Android-Geräte simulieren („emulieren") können, ohne sie tatsächlich als echtes Gerät zu besitzen.

Unnötig zu erwähnen, dass wir Ihnen die wichtigsten Hilfsprogramme in diesem Buch vorstellen und Sie in die Arbeit mit ihnen einführen.

- *Wissen um den Aufbau von Apps und Kenntnis der Android-Klassenbibliothek*
  Dies ist das eigentliche Thema dieses Buchs. Wie sieht das Grundgerüst einer Android-App aus, worauf muss ich achten und was für tolle Sachen kann man mit der Android-Klassenbibliothek machen? (Kurzantwort: Nichts ist unmöglich!)

Nach dem erfolgreichen Durcharbeiten dieses Buchs werden Sie sicher noch kein Profi-Android-Entwickler sein. Das können und wollen wir Ihnen gar nicht versprechen, denn der Umfang an Material wäre so groß, dass kein Platz mehr für ausführliche Erläuterungen bliebe.

Sie werden aber eine sehr fundierte Grundlage erhalten, in viele fortgeschrittene Bereiche blicken und alles Notwendige lernen, um tolle Apps erstellen und sich selbstständig weiterbilden zu können.

Der ideale Leser, Java-Kenntnisse und das Java-Tutorium

Da es den idealen Leser im Grunde gar nicht gibt, sollten wir uns lieber fragen, welche Lesergruppen in welchem Umfang von dem vorliegenden Buch profitieren können:

Leser mit guten Java-Kenntnissen, die sicher objektorientiert programmieren können und bereits Erfahrung mit Konzepten wie Überschreibung, Interface-Implementierung, Ereignis-Listener und Threads haben, bilden eine der drei Hauptzielgruppen, für die dieses Buch geschrieben wurde. Sollten Sie zu dieser Gruppe zählen, legen Sie einfach los.

Leser mit grundlegenden Java-Kenntnissen bilden die zweite Hauptzielgruppe und sollten mit diesem Buch ebenfalls gut und schnell vorankommen. Sollten Sie zu dieser Gruppe gehören, achten Sie auf die im Buchtext eingestreuten Hinweise zu den Exkursen des Java-Tutoriums unter *http://files.hanser.de/fachbuch/PDFs.zip*. Mithilfe dieser Exkurse können Sie etwaige Wissenslücken zur Java-Programmierung schließen.

Umsteiger von anderen Programmiersprachen bilden die dritte Hauptzielgruppe. Doch Obacht! Es liegt viel Arbeit vor Ihnen, denn Sie müssen sich parallel auch noch mithilfe des Java-Tutoriums in Java einarbeiten. Sofern Sie allerdings bereits über gute Programmierkenntnisse in einer anderen objektorientierten Sprache (wie z. B. C++ oder C#) verfügen, dürfte dies für Sie keine große Schwierigkeit sein. Sie können das Tutorium vorab oder parallel zu diesem Buch lesen (die ersten Kapitel enthalten zu diesem Zweck Hinweise, wann Sie welche Teile des Tutoriums lesen sollten).

Bleibt die Gruppe der Leser, die über keine oder nur wenig Programmiererfahrung verfügen. Angehörigen dieser Gruppe können wir eigentlich nur empfehlen, sich zuerst einmal in die Java-Programmierung einzuarbeiten (beispielsweise mit unserem Java-Titel). Sie können es aber natürlich auch mit dem Java-Tutorium versuchen. Es geht zwar relativ flott voran, ist aber recht gut verständlich und beinhaltet sogar eine allgemeine Einführung in die grundlegenden Programmierkonzepte.

Aufbau des Buchs

Das Buch ist in drei Teile plus Anhang gegliedert.

- Der erste Teil behandelt die Installation der notwendigen Entwicklerwerkzeuge und die Grundlagen der App-Erstellung.
- Der zweite Teil vertieft die im ersten Teil angesprochenen Grundthemen: Code, Benutzeroberfläche, Arbeiten mit Ressourcen und der App-Lebenszyklus.
- Der dritte Teil behandelt zahlreiche fortgeschrittene Aspekte wie z. B. Grafik, Menüs, Sensoren, Spiele, Datenbanken oder Geolokation. Er unterscheidet sich nicht nur inhaltlich, sondern auch konzeptionell von den beiden vorangehenden Teilen und ist eher im Stile eines Fortgeschrittenenbuchs geschrieben.

Abgerundet wird das Buch mit Anhängen zur Veröffentlichung von Apps, zur Entwicklungsumgebung Android Studio sowie zu weiteren Werkzeugen wie Emulator, ADM und Debugger, einem Glossar und einem ausführlichen Index.

Software, Beispiel-Apps und sonstiges Material zum Buch

Die Android-Enwicklungsumgebung, die Beispielsammlung und die Tutorials stehen für Sie zum Download bereit. Die Download-Links finden Sie in Anhang D: Das Material zum Buch. Bitte beachten Sie, dass es für die Android-Entwicklungsumgebung und den Java-SDK mehrere Download-Links gibt. Wählen Sie einfach den Link, der zu Ihrem Betriebssystem passt.

Die Website zum Buch

Wir haben dieses Buch mit großer Sorgfalt erstellt. Falls Sie auf Probleme oder Fehler stoßen, sollten Sie nicht zögern, uns eine E-Mail unter Angabe von Buchtitel und Auflage zu senden. Schauen Sie auch einmal auf unserer Buch-Website

www.carpelibrum.de

nach. Neben zusätzlichem Material, den Lösungsprojekten zu den Übungen, Aktualisierungen und Errata finden Sie dort auch weitere Bücher zum Thema Programmieren in Java, C++, C# u. a.

Viel Spaß in der Android-Welt wünschen Ihnen

Dirk Louis (autoren@carpelibrum.de)

Peter Müller (leserfragen@gmx.de)

9 In Views zeichnen

Der Einsatz von UI-Elementen ist eine Möglichkeit, eine App-Oberfläche zu gestalten. Die andere Möglichkeit ist das direkte Zeichnen in die App-Oberfläche oder gezeichnete Figuren (Sprites) vor einem Hintergrund zu bewegen.

9.1 Das Grundprinzip

Um in eine App zeichnen zu können, brauchen wir

- eine *Leinwand* – sprich ein Objekt, in das wir zeichnen können,
- ein *Atelier* – sprich einen Ort, wo wir zeichnen können,
- *Zeichenwerkzeuge* – also Pinsel und Farben.

9.1.1 Die Leinwand

Leinwände sind in der Android-Programmierung Instanzen der Klasse `Canvas`. Jede View verfügt über eine solche `Canvas`-Leinwand, weswegen wir grundsätzlich auch jede View zum Zeichnen verwenden können. Ideal aber sind natürlich Views, die uns eine unverbrauchte, freie Zeichenfläche bieten, wie z. B. die Basisklasse `View` selbst, `ImageView` oder `SurfaceView`.

9.1.2 Das Atelier

Jede View verfügt über eine Methode `onDraw()`, die automatisch vom Android-System aufgerufen wird, wenn die View sichtbar wird und sich selbst zeichnen soll.

Grundsätzlich sollten Sie Ihren Zeichencode immer in diese Methode packen. Erstens müssen Sie sich dann nicht selbst darum kümmern, dass Sie Zugriff auf das `Canvas`-Objekt der View bekommen (dieses wird Ihnen über den Parameter der `onDraw()`-Methode zur Verfü-

gung gestellt). Zweitens ist auf diese Weise sichergestellt, dass Ihr Zeichencode wenn nötig automatisch ausgeführt wird.

Um Zeichencode in die `onDraw()`-Methode einfügen zu können, müssen Sie die Methode allerdings überschreiben – was wiederum bedeutet, dass Sie eine eigene View-Klasse ableiten müssen (vorzugsweise von `View`, `ImageView` oder `SurfaceView`).

Mögliche Alternativen sind das Zeichnen mit `Drawable`-Objekten (für einfache Zeichenoperationen) oder das Zeichnen in eine `SurfaceView` bei gleichzeitiger Implementierung des Interface `SurfaceHolder.Callback`. Auf diese beiden Alternativen werden wir hier allerdings nicht weiter eingehen.

9.1.3 Die Zeichenmethoden und -werkzeuge

Das `Canvas`-Objekt repräsentiert für uns nicht nur die Leinwand, in die wir zeichnen, es stellt uns auch gleich die Methoden zur Verfügung, die wir zum Zeichnen von Linien, Rechtecken, Ovalen, Strings, Bildern und anderen Grafikprimitiven benötigen.

Unterstützt wird die Klasse `android.graphics.Canvas` dabei von der Klasse `android.graphics.Paint`, deren Objekte unsere Zeichenwerkzeuge repräsentieren und über die wir Farbe, Linienbreite und andere Parameter einstellen können.

9.1.4 Wie alles zusammenwirkt

Um einer Bildschirmseite eine View hinzuzufügen, in die Sie zeichnen können, müssen Sie zuerst Ihrem App-Projekt eine neue Quelldatei für Ihre abgeleitete View-Klasse hinzufügen. Danach bauen Sie die View in das Layout der Bildschirmseite ein und zum guten Schluss erweitern Sie die Definition der View-Klasse um Ihren Zeichencode.

Wenn Sie alles gleich parallel zum Lesen in Android Studio ausprobieren wollen, dann wäre es ein guter Zeitpunkt, um ein neues Projekt anzulegen. Verwenden Sie folgende Einstellungen:

Tabelle 9.1 Parameter für das Projekt Grafik

Dialogfeld	Eingabe/Einstellung
Application name	Grafik
Domain name	standard.example.com
Target Android Device	Phone and Tablet, API 15
ADD AN ACtivity	Empty Activity

Quelldatei anlegen

Am besten rufen Sie dazu in der Projektansicht unter dem *app/src/java*-Knoten das Kontextmenü im Paketknoten *com.example.standard.grafik* auf und wählen den Befehl **New/Java Class**. Geben Sie dann den Namen für die neu anzulegende Klasse ein, wie z. B. `ZeichnenView`. Nach dem Erzeugen wird der Code von Android Studio im Editor angezeigt. Wir müssen nun noch von der Basisklasse `android.widget.ImageView` ableiten, indem eine `extends`-Anweisung hinzugefügt und ein Default-Konstruktor bereitgestellt wird (Letzteres können Sie bequem mit **(Alt)+(Einf)** generieren lassen:

Listing 9.1 Anlegen einer Quelldatei für eine View, in die man zeichnen kann

```
package com.example.standard.grafik;

import android.content.Context;
import android.widget.ImageView;

public class ZeichnenView extends ImageView {
    public ZeichnenView(Context context) {
        super(context);
    }
}
```

Grundsätzlich gibt es zwei Techniken, wie Sie Ihre Zeichenflächen-View in eine Bildschirmseite einbauen können:

- Sie bauen die View direkt in den XML-Code der Layoutdatei ein (im Ordner *app/res/layout* zu finden).
- Sie erzeugen die View programmatisch per Java-Code und bauen sie zur Laufzeit in das Layout ein.

Variante 1: Eigene View-Klasse in XML-Layout verwenden

Im ersten Fall müssen Sie in die XML-Layoutdatei ein XML-Element einbauen, das die View repräsentiert. Da die View von einem selbst geschriebenen Typ ist, gibt es kein vordefiniertes XML-Element für diese View (und es taucht daher auch nicht in der **Design**-Ansicht in der Palette auf). Als Name des XML-Elements benutzen Sie daher den vollständigen Namen der View-Klasse (im Beispiel `com.example.standard.grafik.ZeichnenView`).

Listing 9.2 Layoutdatei activity_main.xml mit eigener View-Klasse

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.standard.grafik.MainActivity">

    <com.example.standard.grafik.ZeichnenView
        android:id="@+id/zeichnen"
```

```
        android:layout_width="match_parent"
        android:layout_height="250dp"
        android:background="#ffff00"
        />

</RelativeLayout>
```

Bei Ausführung der App wird für jedes in der Layoutdatei definierte XML-Element ein Objekt der zugehörigen Klasse erzeugt.

Bisher mussten wir uns um diesen Mechanismus nie weiter kümmern, weil er automatisch ablief und wir nur XML-Elemente zu vordefinierten Android-Klassen benutzten.

Wir können diesen Mechanismus auch für selbst geschriebene View-Klassen verwenden, müssen dann aber in der Klasse einen Konstruktor definieren, der als Parameter sowohl ein `Context`-Objekt als auch ein `AttributeSet`-Objekt (zur Übergabe der XML-Attribute) definiert:

```
package com.example.standard.grafik;

import android.content.Context;
import android.util.AttributeSet;
import android.widget.ImageView;

public class ZeichnenView extends ImageView {

    // wenn Klasse programmatisch in der Activity erzeugt wird
    public ZeichnenView(Context context) {
        super(context);
    }

    // wenn Klasse in XML-Layoutdatei verwendet wird
    public ZeichnenView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

ACHTUNG

Wenn Sie keine Hintergrund-Bitmap in die View laden, sollten Sie im XML-Code eine definierte Breite und Höhe vorgeben (also nicht `wrap_content` als Wert verwenden).

Das via XML-Layout erzeugte Objekt können wir uns (wie wir es bisher in den vorangegangenen Kapiteln auch immer gemacht haben) in der `onCreate()`-Methode besorgen und für späteren Gebrauch in einer Variablen merken.

```
public class MainActivity extends AppCompatActivity {
   private ZeichnenView zview;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        zview = (ZeichnenView) findViewById(R.id.zeichnen);
    }
}
```

Variante 2: Eigene View-Klasse programmatisch erzeugen

Wenn man die View-Klasse zur Laufzeit erzeugen will, dann ist die `onCreate()`-Methode der Activity (`MainActivity`) der richtige Platz. Dort erzeugen Sie ein Objekt Ihrer View (in unserem Fall also `ZeichenView`), beschaffen sich einen Verweis auf die gewünschte Layout-View, wo sie eingebaut werden soll, und benutzen deren `addView()`-Methode, um Ihre View als letztes Element in die Layout-View einzufügen.

```
package com.example.standard.grafik;

import android.graphics.Color;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.RelativeLayout;

public class MainActivity extends AppCompatActivity {
 private ZeichnenView zview;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

 zview = new ZeichnenView(this);
 RelativeLayout.LayoutParams params = new
                        RelativeLayout.LayoutParams(
                   RelativeLayout.LayoutParams.MATCH_PARENT, 250);
 zview.setLayoutParams(params);

 zview.setBackgroundColor(Color.YELLOW);

 RelativeLayout layout = (RelativeLayout)findViewById(R.id.layout1);
 layout.addView(zview);
 }
}
```

Die ID für die Layout-View müssen Sie eigenhändig in der XML-Layoutdatei vergeben, da Android Studio keine automatisch generiert, z. B.

```
<RelativeLayout xmlns:... android:id="@+id/layout1" > ...
```

Vergessen Sie übrigens nicht das +-Zeichen, da eine neue ID definiert wird (bei späterem Verweis bzw. Gebrauch der ID im XML-Code muss dann auf das + verzichtet werden).

Zeichencode vorsehen

Um nun in die View zeichnen zu können, überschreiben Sie in Ihrer abgeleiteten View-Klasse die geerbte `onDraw()`-Methode. Sie können das Methodengerüst per Hand anlegen oder einfach in den Code der Klasse klicken, die Tastenkombination **Alt+Einf** drücken (oder

mit rechter Maustaste das Kontextmenü aufrufen und **Generate** auswählen) und dann den Eintrag **Override Methods** markieren. Suchen Sie dann die Methode `onDraw()` und klicken Sie auf **OK**.

Listing 9.3 Überschreiben der `onDraw()`-Methode

```
package com.example.standard.grafik;

import android.content.Context;
import android.graphics.Canvas;
import android.util.AttributeSet;
import android.widget.ImageView;

public class ZeichnenView extends ImageView {
    public ZeichnenView(Context context) {
        super(context);
    }

    public ZeichnenView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
    }
}
```

9.2 Grafikprimitive zeichnen

Das eigentliche Zeichnen geschieht durch Aufruf der dafür vorgesehenen `Canvas`-Methoden.

Zuerst erzeugen Sie ein `Paint`-Objekt, über das Sie z. B. Farbe und gewünschte Linienbreite (für Linien und Umrisse) festlegen.

Dann wählen Sie abhängig von der Art des Grafikprimitivs (Linie, Rechteck, Kreis etc.), das Sie zeichnen möchten, die passende `Canvas`-Methode und schließlich übergeben Sie der Methode die Koordinaten und das `Paint`-Objekt, das die Methode zum Zeichnen des gewünschten Grafikprimitivs verwenden soll.

Listing 9.4 Überschreiben der `onDraw()`-Methode

```
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // 1. Paint-Objekt für Blauton und
   // Linienbreite von 5 Pixeln erzeugen
   Paint pinsel = new Paint();
   pinsel.setColor(Color.rgb(64, 64, 255));
   pinsel.setStrokeWidth(5);
```

```
    // Diagonale durch Leinwand zeichnen
    canvas.drawLine(0, 0, getWidth(), getHeight(), pinsel);
}
```

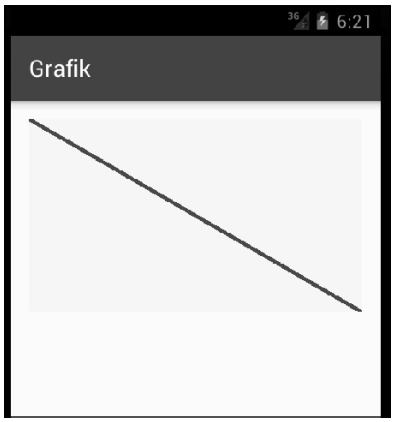

Bild 9.1
Bildschirmseite mit ImageView für eigene Grafikausgaben

Tabelle 9.2 Auswahl einiger Methoden der Klasse `Canvas`

Methode	Beschreibung
`drawColor(int farbe)` `drawRGB(int r,int g,int b)`	Füllt die ganze Leinwand in der angegebenen Farbe (`Color`-Wert oder RGB-Anteile)

`drawArc(RectF r,`
`float start,`
`float winkel,`
`bool keil,`
`Paint pinsel)`
	Zeichnet einen Bogen in das Rechteck `r` ein. Das Argument `start` gibt den Startwinkel an. Über das boolesche Argument können Sie festlegen, ob Sie nur einen Bogen (`false`) oder einen Keil (`true`) zeichnen möchten.

`drawBitmap(Bitmap b,`
`float links,`
`float oben,`
`Paint pinsel)`
	Kopiert ein Bild in das Canvas, sodass die linke obere Ecke des Bilds an der angegebenen Koordinate zu liegen kommt. Für das `Paint`-Objekt können Sie `null` übergeben oder ein Objekt, das eine Maske für die Verblendung mit dem Hintergrund definiert.

`drawBitmap(Bitmap b,`
`RectF quelle,`
`RectF ziel,`
`Paint pinsel)`
	Kopiert ein Bild oder einen Bildausschnitt (Argument `quelle` ungleich `null`) in den angegebenen Zielbereich des Canvas. Das Bild wird notfalls skaliert, damit es in den Zielbereich passt. Für das `Paint`-Objekt können Sie `null` übergeben oder ein Objekt, das eine Maske für die Verblendung mit dem Hintergrund definiert.

`drawBitmap(Bitmap b, Matrix matrix,` `Paint pinsel)`	Kopiert das Bild an die durch eine Transformationsmatrix angegebene Stelle. `matrix` kann eine beliebige Kombination von Verschiebung/Rotation/Skalierung beinhalten.

(Fortsetzung nächste Seite)

Tabelle 9.2 Auswahl einiger Methoden der Klasse `Canvas` *(Fortsetzung)*

Methode	Beschreibung
`drawCircle(float x, float y, float radius, Paint pinsel)`	Zeichnet einen Kreis mit Mittelpunkt x,y und dem angegebenen Radius.
`drawLine(float startX, float startY, float endX, float endY, Paint pinsel)`	Zeichnet eine Linie von (`startX`,`startY`) nach (`endX`,`endY`).
`drawLines(float[] punkte, Paint pinsel)`	Zeichnet mehrere Linien. Für jede Linie müssen im Array `punkte` vier Werte angegeben werden.
`drawOval(RectF r, Paint pinsel)`	Zeichnet ein Oval in das vorgegebene Rechteck. Wenn Sie als erstes Argument ein Quadrat übergeben, erhalten Sie einen Kreis.
`drawPoints(float[] punkte, int offset, int anzahl, Paint pinsel)`	Zeichnet mehrere Punkte. Für jeden Punkt müssen im Array `punkte` zwei Werte (x,y) angegeben werden. Die ersten `offset` Punkte werden übersprungen und es werden maximal `anzahl` Punkte gezeichnet. Die Dicke der Punkte können Sie über den Stroke-Wert des Pinsels festlegen.
`drawRect(float links, float oben, float rechts, float unten, Paint pinsel)` `drawRect(RectF r, Paint pinsel)`	Zeichnet ein Rechteck.
`drawRoundRect(RectF r, float rx, float ry, Paint pinsel)`	Zeichnet ein Rechteck mit abgerundeten Ecken. Die Radien für die abgerundeten Ecken können Sie über die Argumente `rx` und `ry` vorgeben.
`drawText(String text, float x, float y, Paint pinsel)`	Zeichnet einen String an der spezifizierten Koordinate.
`fillArc(), fillOval(), fillPolygon()` ...	Zeichnet ausgefüllte Formen (vgl. `draw...()`).
`scale(float fx, float fy)`	Skaliert das Canvas um den Faktor fx (für x-Achse) und fy (y-Achse).
`translate(float dx, float dy)`	Verschiebt den Ursprung des Canvas.

Koordinaten

Koordinaten können Sie als `int`- oder `float`-Werte angeben.

Etliche Methoden arbeiten mit umschließenden Rechtecken (Bounding Box), in die das zu zeichnende Grafikprimitiv eingepasst wird. Diese Rechtecke können Sie meist auch als `RectF`-Objekt angeben.

```
Paint pinsel = new Paint();
pinsel.setColor(Color.CYAN);
pinsel.setStrokeWidth(2);

RectF r = new RectF(10, 10, 50, 50);
canvas.drawRect(r, pinsel);
```

Der Ursprung des Canvas liegt immer in der linken oberen Ecke (Koordinaten = 0,0) und die x-Werte nehmen nach rechts hin zu, die y-Werte nach unten hin.

Farben

Farben werden grundsätzlich durch `int`-Werte codiert. Einige Farben sind als statische Felder der Klasse `android.graphics.Color` vordefiniert: `Color.BLACK`, `Color.BLUE`, `Color.CYAN`, `Color.DKGRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LTGRAY`, `Color.MAGENTA`, `Color.RED`, `Color.TRANSPARENT`, `Color.WHITE`, `Color.YELLOW`. Alle anderen Farben können Sie mithilfe der statischen Methode `rgb()` definieren:

```
Paint pinsel = new Paint();
pinsel.setColor(Color.rgb(64, 64, 255));
pinsel.setStrokeWidth(2);

RectF r = new RectF(10, 10, 50, 50);
canvas.drawRect(r, pinsel);
```

Umrisse oder gefüllte Primitive

Mit den `draw`-Methoden können Sie sowohl Umrisslinien als auch ausgefüllte Grafikprimitive zeichnen. Standardmäßig werden ausgefüllte Grafikprimitive gezeichnet (`Style.FILL`). Um einen Umriss zu zeichnen, müssen Sie die `setStyle()`-Methode des verwendeten `Paint`-Objekts aufrufen und dieser die Konstante `Style.STROKE` übergeben.

```
RectF r = new RectF(10, 10, 50, 50);

Paint umrissPinsel = new Paint();
umrissPinsel.setColor(Color.YELLOW);
umrissPinsel.setStrokeWidth(2);
umrissPinsel.setStyle(Style.STROKE);

Paint fuellPinsel = new Paint();
fuellPinsel.setColor(Color.MAGENTA);
fuellPinsel.setStyle(Style.FILL);

canvas.drawRect(r, fuellPinsel);
canvas.drawRect(r, umrissPinsel);
```

9.3 Bilder laden

Wenn Sie beispielsweise in einer View ein Hintergrundbild setzen wollen, dann reicht es, einen Aufruf der Art `setBackgroundResource(R.drawable.meinbild)` zu machen, und das war es schon. Android macht alle notwendigen Dinge.

Um eine Bildressource in einer App selbst zu verwenden, muss man leider etwas mehr Aufwand treiben, insbesondere, wenn man nicht weiß oder sicherstellen kann, dass das zu ladende Bild recht groß ist (einige Mbyte Platzbedarf). In solchen Fällen kommt es sehr schnell zu OutOfMemory-Fehlern, weil der verfügbare Speicher nicht ausreicht.

Das Laden von Bildern, die groß sind oder wo Sie es nicht wissen (können), sollten Sie daher immer in drei Schritten ausführen:

- Ermitteln Sie die Größe (Breite/Höhe) des zu ladenden Bildes. Dies erfordert ein provisorisches Laden des Bildes ohne die eigentlichen Bilddaten.
- Berechnen Sie einen geeigneten Skalierungsfaktor, damit es für die geplante Anzeige (eine View o. Ä.) passt.
- Laden Sie das Bild.

Codemäßig sehen die Schritte dann folgendermaßen aus. Zuerst lädt man das Bild mit der statischen Methode `BitmapFactory.decodeResource()` und übergibt dabei ein `BitmapFactory.Options`-Objekt, dessen Feld `inJustDecodeBounds` auf `true` gesetzt ist:

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), id, options);
```

Das `options`-Objekt enthält nun die Bildmaße in den Variablen `outWidth` und `outHeight`. Dann muss man den Anpassungsfaktor berechnen. Man startet mit 1 und, falls das Bild größer ist als der geplante Anzeigeort, wird er immer weiter verdoppelt, bis das Verhältnis Größe/Faktor klein genug geworden ist:

```
private int berechneLadefaktor(BitmapFactory.Options options,
                               int breiteAnzeige, int hoeheAnzeige){
final int w = options.outWidth;
final int h = options.outHeight;
int faktor = 1;

if (w > breiteAnzeige || h > hoeheAnzeige) {
  int w_halbe = w / 2;
  int h_halbe = h / 2;

  while((w_halbe / faktor) > breiteAnzeige
        && ((h_halbe / faktor) > hoeheAnzeige)) {
        faktor = faktor * 2;
      }
    }

return faktor;
}
```

Nun kann man die Bilddaten laden. Wichtig ist dabei, den berechneten Faktor an die Variable `inSampleSize` zuzuweisen und das Flag `inJustDecodeBounds` wieder auf `false` zu stellen, z. B.

```
options.inSampleSize = berechneLadefaktor(options,
                                          viewBreite, viewHoehe);
options.inJustDecodeBounds = false;
Bitmap bild = BitmapFactory.decodeResource(getResources(),
                                    R.drawable.meinbild, options);
```

9.4 In Bilder hineinzeichnen

Bilder, die man als `Bitmap`-Objekte geladen hat, kann man nicht einfach nur anzeigen. Man kann sie auch mit eigenen Zeichenoperationen verändern. Das ist oft einfacher und vom Ergebnis her viel ansprechender, als alles komplett mithilfe der Zeichenmethoden der Klasse `Canvas` selbst zu erstellen. Hierzu muss man einfach ein eigenes `Canvas`-Objekt erzeugen, wobei man dem Konstruktor das Bitmap mitgibt. Allerdings lauert hier eine kleine Falle: Wenn das `Bitmap`-Objekt durch das Laden einer Bildressource erzeugt worden ist, dann es ist nicht veränderbar *(immutable)*. Man muss daher zuvor eine Kopie anlegen und kann dann darauf ein `Canvas`-Objekt erstellen:

```
BitmapFactory.Options options = new BitmapFactory.Options();
Bitmap origBild = BitmapFactory.decodeResource(getResources(),
                                    R.drawable.meinbild, options);
// Kopie erzeugen
Bitmap kopie = origBild.copy(origBild.getConfig(), true);

// Canvas zum Zeichnen
Canvas canvasKopie = new Canvas(kopie);
canvasKopie.drawRect(...) // usw.
```

Nun hat man ein `Canvas`-Objekt und kann auf der Bildkopie zeichnen. Alle Veränderungen werden sichtbar, wenn das `Bitmap`-Objekt (hier `kopie`) via `drawImage()` in einer `View.onDraw()`-Methode gerendert wird, z. B.

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawBitmap(kopie, 0, 0, null);
}
```

Als kleine Veranschaulichung finden Sie unter *http://files.hanser.de/fachbuch/Beispiele.zip* zu diesem Kapitel das Beispielprojekt *Kratzbild*. Hierbei wird mit zwei Bitmaps gearbeitet: Eins dient zur Darstellung eines Fotos, das andere ist zu Beginn einfach schwarz und verdeckt das Foto. Der Benutzer kann es durch Fingerbewegung freikratzen und das Foto kommt nach und nach zum Vorschein.

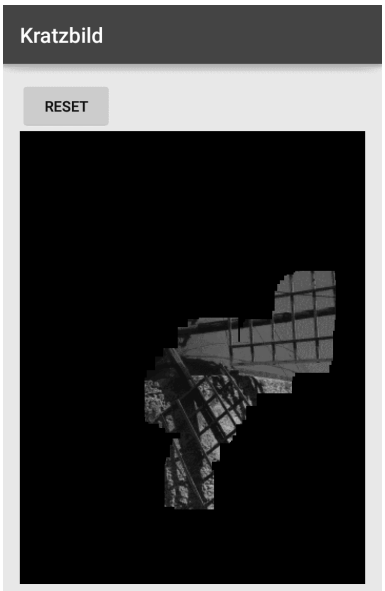

Bild 9.2
Beispiel-App Kratzbild

Wir betrachten nur die zwei bzgl. des Zeichnens interessanten Codeteile.

Zum einen wird eine neue Bitmap mithilfe der statischen Methode `Bitmap.createBitmap()` in der Größe der Anzeigefläche erstellt. Auf dieser Bitmap erzeugen wir nun ein `Canvas`-Objekt und zeichnen darin mit einem `drawRect()`-Befehl ein alles bedeckendes schwarzes Rechteck:

```
Bitmap kratzSchicht =  Bitmap.createBitmap(anzeigeBreite,
                           anzeigeHoehe, Bitmap.Config.ARGB_8888);
Canvas kratzSchichtCanvas = new Canvas(kratzSchicht);
Rect r = new Rect(0,0, kratzSchicht.getWidth(),
                       kratzSchicht.getHeight());
Paint paint = new Paint();
paint.setColor(Color.BLACK);
kratzSchichtCanvas.drawRect(r, paint);
```

In der `onDraw()`-Methode der View wird immer zuerst das anzuzeigende Bild (Bitmap `kratzbild`) gezeichnet und dann darüber die schwarze Kratzschicht. Zu Beginn sieht der Benutzer also nur schwarz:

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

   // ...
    canvas.drawBitmap(kratzbild, offsetX, offsetY, null);

    canvas.drawBitmap(kratzSchicht, 0, 0, null);
}
```

Wenn der Benutzer nun mit dem Finger herumkratzt, fangen wir die Mausbewegungen ab und machen an dieser Stelle die Kratzschicht durchsichtig, d. h., wir müssen dem Canvas `kratzschichtCanvas` den Befehl erteilen, an einer bestimmten Stelle schwarz durch transparent zu ersetzen. Hierzu brauchen wir ein `Paint`-Objekt mit Transfer-Modus[1] CLEAR:

```
Paint transparentPaint = new Paint();
transparentPaint.setXfermode(new
                        PorterDuffXfermode(PorterDuff.Mode.CLEAR));
```

Bei jedem Mausevent müssen wir dann nur noch auf der `kratzschichtCanvas` an der aktuellen Stelle ein „Loch brennen“ und per `invalidate()` das Neuzeichnen veranlassen.

```
public void updateMouse(int x, int y) {
  kratzSchichtCanvas.drawRect(x, y, x + 75, y + 75,
                              transparentPaint);
  invalidate();
}
```

9.5 Bilder bewegen

In diesem Abschnitt möchten wir Ihnen zeigen, wie man es einrichten kann, dass der Anwender eine Figur (ein UFO) mithilfe der Finger über den App-Hintergrund bewegt.

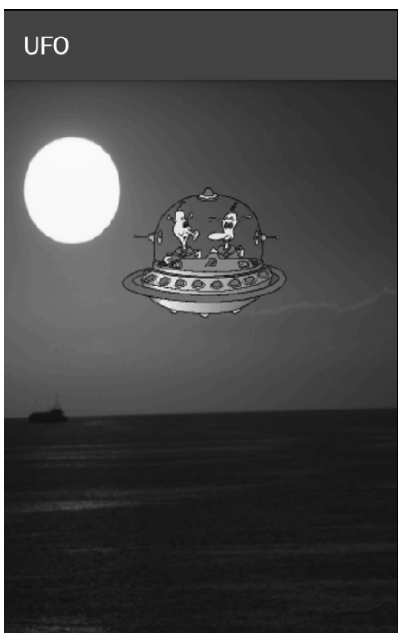

Bild 9.3
Eine fliegende Untertasse

[1] Leider sprengt die Komplexität von Computergrafik den Rahmen dieses Buches, um dies und vieles anderes weiter zu erläutern.

Dazu benötigen wir:

- eine App mit einem `FrameLayout` und einer selbst definierten View
- eine Bitmap für das UFO vor transparentem Hintergrund
- Code zum Zeichnen des UFO
- Code zur Behandlung der Richtung

Falls Sie die Schritte parallel zum Lesen in Android Studio nachverfolgen wollen, dann sollten Sie ein neues Projekt mit den üblichen Einstellungen anlegen:

Tabelle 9.3 Parameter des Projekts UFO

Dialogfeld	Eingabe/Einstellung
Application name	UFO
Domain name	standard.example.com
Target Android Device	Phone and Tablet, API 15
ADD AN ACtivity	Empty Activity

View-Klasse, Activity und Layoutdatei

In Android Studio neu angelegte App-Projekte verfügen (bei Erstellung einer Empty Activity) immer über eine Start-Activity mit einem `RelativeLayout`. Für Apps, die wie unsere UFO-App nur aus einer einzigen Zeichenfläche bestehen (also einem einzigen anzuzeigenden View-Element), ist das `FrameLayout` besser geeignet. Der erste Schritt besteht also darin, das `RelativeLayout` durch `FrameLayout` zu ersetzen.

Listing 9.5 Die Layoutdatei activity_main.xml (aus dem Projekt UFO)

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:id="@+id/framelayout0"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
</FrameLayout>
```

Um die oberste Layout-View auszutauschen, öffnen Sie die Layoutdatei *activity_main.xml* am besten in der XML-Ansicht.

Als Zeichenfläche verwenden wir eine selbst geschriebene View-Klasse, die wir direkt von der Basisklasse `View` ableiten. Im Konstruktor dieser Klasse laden wir die Bildressource für den Hintergrund. Hierzu haben wir ein geeignetes Foto als Datei *hintergrund.png* im Projektordner *app\src\main\res\drawable* abgelegt.

Listing 9.6 Definition der Zeichenflächen-View (aus UFOView.java)

```
package com.standard.example.ufo;

import android.content.Context;
import android.content.res.Resources;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.view.MotionEvent;
import android.view.View;


public class UFOView extends View  {
   public UFOView(Context c) {
      super(c);

      // lade Hintergrund
      this.setBackgroundResource(R.drawable.hintergrund);
   }
}
```

In der `onCreate()`-Methode der Activity-Klasse erzeugen wir nun ein Objekt unserer View-Klasse und fügen dieses programmatisch durch Java-Code zur Laufzeit in das `FrameLayout` ein.

Listing 9.7 Einbetten der Zeichenflächen-View (aus MainActivity.java)

```
package com.standard.example.ufo;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.FrameLayout;

public class MainActivity extends AppCompatActivity {
    private UFOView ufoView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      // UI vervollständigen
      ufoView = new UFOView(this);

      FrameLayout frameLayout =
                     (FrameLayout)findViewById(R.id.framelayout1);
      frameLayout.addView(ufoView);
    }
}
```

Die UFO-Bitmap

Bitmaps haben immer rechteckige Abmaße. Wenn Sie daher Bitmaps für Figuren erzeugen, die Sie über einen Hintergrund oder zwischen anderen Figuren bewegen möchten, müssen

Sie darauf achten, dass Sie die Figur vor einem einfarbigen Hintergrund zeichnen und diesen dann als transparent deklarieren. Mit etwas besser ausgestatteten Grafikprogrammen sollte dies kein Problem sein.

SPRITES

Figuren, die bewegt werden, bezeichnet man auch als *Sprites*. Speichern Sie Ihre Sprites im PNG-Format. (Das JPG-Format unterstützt keine Transparenz.)

Speichern Sie die Bitmap der Figur als Ressource *ufo.png* (Ordner *app\src\main\res\drawable)*. Laden Sie die Bitmap-Ressource in der `onCreate()`-Methode der View-Klasse in ein `Bitmap`-Objekt. Verwenden Sie dazu die `BitmapFactory`-Methode `decodeResource()`.

```
public class UFOView extends View  {
   private float xpos = -1;
   private float ypos = -1;
   private Bitmap ufoBitmap;

   public UFOView(Context c) {
      super(c);

      // lade Hintergrund
      this.setBackgroundResource(R.drawable.hintergrund);

      // lade Bitmap für UFO
      Resources resources = getResources();
      ufoBitmap = BitmapFactory.decodeResource(resources,
                                               R.drawable.ufo);
   }
}
```

Haben Sie die Felder `xpos` und `ypos` bemerkt? Sie sollen die Position (Mittelpunkt) angeben, an der die UFO-Bitmap in das Canvas gezeichnet werden soll.

Anfangs haben diese Koordinaten den Wert –1, obwohl wir das UFO eigentlich in der Mitte der View anzeigen möchten. Die Methoden, die uns Breite und Höhe der View liefern (`getWidth()` und `getHeight()`), geben allerdings in der `onCreate()`-Methode noch keine vernünftigen Werte zurück (auch nicht in `onStart()`). Wir starten daher mit den Anfangswerten –1 und korrigieren diese beim ersten Aufruf von `onDraw()`. Später werden die Koordinaten dann nur noch beim Tippen durch den Anwender verändert.

onDraw()

In der `onDraw()`-Methode kopieren wir die Bitmap in das Canvas. Als Zielposition wählen wir die Koordinate (`xpos`, `ypos`), wobei wir von dem x-Wert die halbe Bitmap-Breite und von dem y-Wert die halbe Bitmap-Höhe abziehen, damit die Bitmap ungefähr mittig über dem Punkt (`xpos`, `ypos`) zu liegen kommt.

```
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // Startposition beim ersten Aufruf auf View-Mitte setzen
```

```
   if (xpos == -1 && ypos == -1) {
      xpos = getWidth()/2;
      ypos = getHeight()/2;
   }

   // UFO-Bitmap einzeichnen
   Paint iconPaint = new Paint(Paint.ANTI_ALIAS_FLAG |
                               Paint.DITHER_FLAG |
                               Paint.FILTER_BITMAP_FLAG
);

   if(ufoBitmap != null) {
       canvas.drawBitmap(ufoBitmap,
                         xpos - ufoBitmap.getWidth()/2,
                         ypos - ufoBitmap.getHeight()/2,
                         iconPaint);
   }
}
```

Achten Sie insbesondere darauf, dass Sie für das Zeichnen eines Bitmaps ein `Paint`-Objekt benutzen sollten, das mit verschiedenen Schaltern *(Flags)* zum optimierten Zeichnen von Bitmaps initialisiert worden ist (`Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG | Paint.FILTER_BITMAP_FLAG`).

Beim ersten Aufruf von `onDraw()` korrigieren wir die Werte von `xpos` und `ypos` so, dass das UFO in der Mitte der View angezeigt wird.

onTouchEvent()

Um sicherzustellen, dass unsere Zeichenflächen-View überhaupt Ereignisse empfängt, rufen wir in ihrer `onCreate()`-Methode die Methode `setFocusable()` auf:

```
public UFOView(Context c) {
   super(c);

   // ...

   setFocusable(true);
}
```

Der Anwender soll das UFO durch Tippen mit dem Finger steuern. Wir brauchen also eine Behandlung von Touch-Ereignissen. Hierbei haben wir zwei Möglichkeiten:

- Wir registrieren bei der `UFOView` einen eigenen `OnTouchListener` oder
- wir überschreiben die Methode `onTouch()` in der `UFOView`-Klasse

Beide Ansätze sind absolut gleichwertig und wir nehmen mal die zweite Variante und überschreiben die `onTouch()`-Methode:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();

    float dx = event.getX() - xpos;
    float dy = event.getY() - ypos;
```

```
    if(action == MotionEvent.ACTION_MOVE) {
        xpos += dx;
        ypos += dy;
    }
    else  if(action == MotionEvent.ACTION_DOWN ) {
        xpos += Math.signum(dx) * 25;
        ypos += Math.signum(dy) * 25;
    }

    if(xpos < 0) {
        xpos = 0;
    }
    if(xpos > getWidth()) {
        xpos = getWidth();
    }
    if(ypos < 0) {
        ypos = 0;
    }
    if(ypos > getHeight()) {
        ypos = getHeight();
    }

    invalidate();
    return true;
}
```

Wurde eine Berührung (`ACTION_DOWN`) oder ein Wischen des Fingers (`ACTION_MOVE`) ausgeführt, dann ermitteln wir die Distanz zwischen der Stelle, wo der Finger berührt, und der aktuellen Mitte des UFO. Das ergibt die Information, in welche Richtung sich das UFO bewegen soll. Im Falle eines einzelnen Tippens verschieben wir das UFO pauschal um 25 Pixel, bei der Dauerberührung um die aktuelle Pixeldifferenz.

Zum Schluss erfolgt noch eine Prüfung, ob die UFO-Position in Gefahr gerät, den Bildschirm zu verlassen. In solchen Fällen stoppen wir das UFO an der jeweiligen Grenze.

Am Ende der Methode rufen wir `invalidate()` auf. Der `invalidate()`-Aufruf bewirkt, dass die `UFOView` ein Signal erhält, dass sie sich selbst neu zeichnen soll, d. h., es wird im Endeffekt `onDraw()` ausgeführt und dabei werden die neuen Werte von `xpos` und `ypos` verwendet.

9.6 Verbesserungen

Die hier vorgestellten Techniken eignen sich allerdings nur für einfache Grafikanwendungen. Wenn Sie für die Aktualisierung der gezeichneten Oberfläche länger andauernde Berechnungen durchführen müssen oder mehrere Objekte dynamisch und schnell verschieben möchten, lagern Sie den Code zum Zeichnen in einen Thread aus, verwenden Sie `SurfaceView` und `SurfaceHolder.Callback` und führen Sie alle Zeichenoperationen im Hintergrund auf einem eigenen Canvas aus, den Sie dann erst nach Abschluss der Zeichenoperationen in einem Rutsch in den Canvas der View kopieren.

9.7 Fragen und Antworten

1. *Irgendwo habe ich einmal gewundenen Text gesehen. Wie haben die das gemacht?*

   Vermutlich mit der Canvas-Methode `drawTextOnPath()`. Hier wird ein String entlang eines vorgegebenen Pfads (Klasse `android.graphics.Path`) gezeichnet.

2. *Wie kann man halb durchsichtig (transparent) zeichnen?*

   Transparenz wird im RGB-Farbmodell durch den sogenannten Alpha-Kanal festgelegt. Die `Paint`-Klasse bietet daher die Methode `setAlpha()`, die einen Wert zwischen 0 (völlig transparent) und 255 (undurchsichtig) erwartet.

3. *Moderne Programmiersprachen bieten auch Gradientenfüllung. Was ist mit Android?*

   Gibt's natürlich auch. Man kann z. B. die Klasse `LinearGradient` verwenden, um einen linearen Farbverlauf von einem Startpunkt zu einem Endpunkt und von Gelb zu Blau zu erreichen:

```
LinearGradient lg = new LinearGradient(startx, starty,
                                       endx, endy,
                           Color.YELLOW, Color.BLUE, TileMode.CLAMP);

Paint pinsel = new Paint();
pinsel.setShader(lg);
```

9.8 Übung

Schreiben Sie einen Funktionsplotter, der für die Funktion f(x) = sin(x) den Verlauf im Wertebereich – 5 bis + 5 zeichnet.

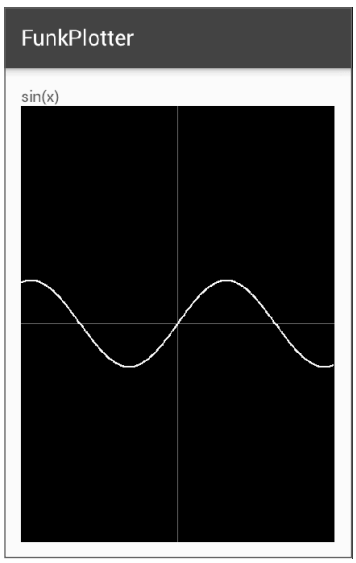

Bild 9.4
Ausgabe von sin(x)

Index

B

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

U

V

W

X

Z